CHAPTER 1

Introduction and Essential Concepts

This book is about system programming, which is the art of writing system software. System software lives at a low level, interfacing directly with the kernel and core system libraries. System software includes your shell and your text editor, your compiler and your debugger, your core utilities and system daemons. These components are entirely system software, based on the kernel and the C library. Much other software (such as high-level GUI applications) lives mostly in the higher levels, delving into the low level only on occasion, if at all. Some programmers spend all day every day writing system software; others spend only part of their time on this task. There is no programmer, however, who does not benefit from some understanding of system programming. Whether it is the programmer’s raison d’être, or merely a foundation for higher-level concepts, system programming is at the heart of all software that we write.

In particular, this book is about system programming on Linux. Linux is a modern Unix-like system, written from scratch by Linus Torvalds, and a loose-knit community of hackers around the globe. Although Linux shares the goals and ideology of Unix, Linux is not Unix. Instead, Linux follows its own course, diverging where desired, and converging only where practical. Generally, the core of Linux system programming is the same as on any other Unix system. Beyond the basics, however, Linux does well to differentiate itself—in comparison with traditional Unix systems, Linux is rife with additional system calls, different behavior, and new features.

System Programming

Traditionally speaking, all Unix programming is system-level programming. Historically, Unix systems did not include many higher-level abstractions. Even programming in a development environment such as the X Window System exposed in full view the core Unix system API. Consequently, it can be said that this book is a book on Linux
programming in general. But note that this book does not cover the Linux programming \textit{environment}—there is no tutorial on \texttt{make} in these pages. What is covered is the system programming API exposed on a modern Linux machine.

System programming is most commonly contrasted with application programming. System-level and application-level programming differ in some aspects, but not in others. System programming is distinct in that system programmers must have a strong awareness of the hardware and operating system on which they are working. Of course, there are also differences between the libraries used and calls made. Depending on the “level” of the stack at which an application is written, the two may not actually be very interchangeable, but, generally speaking, moving from application programming to system programming (or vice versa) is not hard. Even when the application lives very high up the stack, far from the lowest levels of the system, knowledge of system programming is important. And the same good practices are employed in all forms of programming.

The last several years have witnessed a trend in application programming away from system-level programming and toward very high-level development, either through web software (such as JavaScript or PHP), or through managed code (such as C# or Java). This development, however, does not foretell the death of system programming. Indeed, someone still has to write the JavaScript interpreter and the C# runtime, which is itself system programming. Furthermore, the developers writing PHP or Java can still benefit from knowledge of system programming, as an understanding of the core internals allows for better code no matter where in the stack the code is written.

Despite this trend in application programming, the majority of Unix and Linux code is still written at the system level. Much of it is C, and subsists primarily on interfaces provided by the C library and the kernel. This is traditional system programming—Apache, \texttt{bash}, \texttt{cp}, Emacs, \texttt{init}, \texttt{gcc}, \texttt{gdb}, \texttt{glibc}, \texttt{ls}, \texttt{mv}, \texttt{vim}, and \texttt{X}. These applications are not going away anytime soon.

The umbrella of system programming often includes kernel development, or at least device driver writing. But this book, like most texts on system programming, is unconcerned with kernel development. Instead, it focuses on user-space system-level programming; that is, everything above the kernel (although knowledge of kernel internals is a useful adjunct to this text). Likewise, network programming—sockets and such—is not covered in this book. Device driver writing and network programming are large, expansive topics, best tackled in books dedicated to the subject.

What is the system-level interface, and how do I write system-level applications in Linux? What exactly do the kernel and the C library provide? How do I write optimal code, and what tricks does Linux provide? What neat system calls are provided in Linux compared to other Unix variants? How does it all work? Those questions are at the center of this book.

There are three cornerstones to system programming in Linux: system calls, the C library, and the C compiler. Each deserves an introduction.
System Calls

System programming starts with system calls. System calls (often shortened to syscalls) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) in order to request some service or resource from the operating system. System calls range from the familiar, such as `read()` and `write()`, to the exotic, such as `get_thread_area()` and `set_tid_address()`.

Linux implements far fewer system calls than most other operating system kernels. For example, a count of the i386 architecture’s system calls comes in at around 300, compared with the allegedly thousands of system calls on Microsoft Windows. In the Linux kernel, each machine architecture (such as Alpha, i386, or PowerPC) implements its own list of available system calls. Consequently, the system calls available on one architecture may differ from those available on another. Nonetheless, a very large subset of system calls—more than 90 percent—is implemented by all architectures. It is this shared subset, these common interfaces, that I cover in this book.

Invoking system calls

It is not possible to directly link user-space applications with kernel space. For reasons of security and reliability, user-space applications must not be allowed to directly execute kernel code or manipulate kernel data. Instead, the kernel must provide a mechanism by which a user-space application can “signal” the kernel that it wishes to invoke a system call. The application can then trap into the kernel through this well-defined mechanism, and execute only code that the kernel allows it to execute. The exact mechanism varies from architecture to architecture. On i386, for example, a user-space application executes a software interrupt instruction, `int`, with a value of 0x80. This instruction causes a switch into kernel space, the protected realm of the kernel, where the kernel executes a software interrupt handler—and what is the handler for interrupt 0x80? None other than the system call handler!

The application tells the kernel which system call to execute and with what parameters via `machine registers`. System calls are denoted by number, starting at 0. On the i386 architecture, to request system call 5 (which happens to be `open()`), the user-space application stuffs 5 in register eax before issuing the `int` instruction.

Parameter passing is handled in a similar manner. On i386, for example, a register is used for each possible parameter— registers ebx, ecx, edx, esi, and edi contain, in order, the first five parameters. In the rare event of a system call with more than five parameters, a single register is used to point to a buffer in user space where all of the parameters are kept. Of course, most system calls have only a couple of parameters.

Other architectures handle system call invocation differently, although the spirit is the same. As a system programmer, you usually do not need any knowledge of how the kernel handles system call invocation. That knowledge is encoded into the standard calling conventions for the architecture, and handled automatically by the compiler and the C library.
The C Library

The C library (libc) is at the heart of Unix applications. Even when you’re programming in another language, the C library is most likely in play, wrapped by the higher-level libraries, providing core services, and facilitating system call invocation. On modern Linux systems, the C library is provided by GNU libc, abbreviated glibc, and pronounced *gee-lib-see* or, less commonly, *glib-see*.

The GNU C library provides more than its name suggests. In addition to implementing the standard C library, glibc provides wrappers for system calls, threading support, and basic application facilities.

The C Compiler

In Linux, the standard C compiler is provided by the GNU Compiler Collection (gcc). Originally, gcc was GNU’s version of cc, the C Compiler. Thus, gcc stood for GNU C Compiler. Over time, support was added for more and more languages. Consequently, nowadays gcc is used as the generic name for the family of GNU compilers. However, gcc is also the binary used to invoke the C compiler. In this book, when I talk of gcc, I typically mean the program gcc, unless context suggests otherwise.

The compiler used in a Unix system—Linux included—is highly relevant to system programming, as the compiler helps implement the C standard (see “C Language Standards”) and the system ABI (see “APIs and ABIs”), both later in this chapter.

APIs and ABIs

Programmers are naturally interested in ensuring their programs run on all of the systems that they have promised to support, now and in the future. They want to feel secure that programs they write on their Linux distributions will run on other Linux distributions, as well as on other supported Linux architectures and newer (or earlier) Linux versions.

At the system level, there are two separate sets of definitions and descriptions that impact portability. One is the application programming interface (API), and the other is the application binary interface (ABI). Both define and describe the interfaces between different pieces of computer software.

APIs

An API defines the interfaces by which one piece of software communicates with another at the source level. It provides abstraction by providing a standard set of interfaces—usually functions—that one piece of software (typically, although not

necessarily lower-level on the stack) can use in a consistent manner. It is composed of the tools the programmer builds into their programs. At the API level, as part of the build process, the programmer writes the higher-level code, and the API provides the standard implementation that the programmer can rely on to work. Through the use of standard language constructs, a real-world standard is developed.

ABIs

Whereas APIs provide how an API and how to use it, ABIs provide an API with the binary interface. ABIs are used in the build process to make the programs work. The call interfaces are and how to use them. Although ABIs are higher-level efforts that include embedded hardware concepts, architecture, machine instruction sets, and system calls, they provide a consistent way to build programs that run on different systems.
necessarily, a higher-level piece) can invoke from another piece of software (usually a lower-level piece). For example, an API might abstract the concept of drawing text on the screen through a family of functions that provide everything needed to draw the text. The API merely defines the interface; the piece of software that actually provides the API is known as the implementation of the API.

It is common to call an API a “contract.” This is not correct, at least in the legal sense of the term, as an API is not a two-way agreement. The API user (generally, the higher-level software) has zero input into the API and its implementation. It may use the API as-is, or not use it at all: take it or leave it! The API acts only to ensure that if both pieces of software follow the API, they are source compatible; that is, that the user of the API will successfully compile against the implementation of the API.

A real-world example is the API defined by the C standard and implemented by the standard C library. This API defines a family of basic and essential functions, such as string-manipulation routines.

Throughout this book, we will rely on the existence of various APIs, such as the standard I/O library discussed in Chapter 3. The most important APIs in Linux system programming are discussed in the section “Standards” later in this chapter.

**ABIs**

Whereas an API defines a source interface, an ABI defines the low-level binary interface between two or more pieces of software on a particular architecture. It defines how an application interacts with itself, how an application interacts with the kernel, and how an application interacts with libraries. An ABI ensures binary compatibility, guaranteeing that a piece of object code will function on any system with the same ABI, without requiring recompilation.

ABIs are concerned with issues such as calling conventions, byte ordering, register use, system call invocation, linking, library behavior, and the binary object format. The calling convention, for example, defines how functions are invoked, how arguments are passed to functions, which registers are preserved and which are mangled, and how the caller retrieves the return value.

Although several attempts have been made at defining a single ABI for a given architecture across multiple operating systems (particularly for i386 on Unix systems), the efforts have not met with much success. Instead, operating systems—Linux included—tend to define their own ABIs however they see fit. The ABI is intimately tied to the architecture; the vast majority of an ABI speaks of machine-specific concepts, such as particular registers or assembly instructions. Thus, each machine architecture has its own ABI on Linux. In fact, we tend to call a particular ABI by its machine name, such as *alpha*, or *x86-64*.
System programmers ought to be aware of the ABI, but usually do not need to memorize it. The ABI is enforced by the toolchain—the compiler, the linker, and so on—and does not typically otherwise surface. Knowledge of the ABI, however, can lead to more optimal programming, and is required if writing assembly code or hacking on the toolchain itself (which is, after all, system programming).

The ABI for a given architecture on Linux is available on the Internet and implemented by that architecture's toolchain and kernel.

**Standards**

Unix system programming is an old art. The basics of Unix programming have existed untouched for decades. Unix systems, however, are dynamic beasts. Behavior changes and features are added. To help bring order to chaos, standards groups codify system interfaces into official standards. Numerous such standards exist, but technically speaking, Linux does not officially comply with any of them. Instead, Linux aims toward compliance with two of the most important and prevalent standards: POSIX and the Single UNIX Specification (SUS).

POSIX and SUS document, among other things, the C API for a Unix-like operating system interface. Effectively, they define system programming, or at least a common subset thereof, for compliant Unix systems.

**POSIX and SUS History**

In the mid-1980s, the Institute of Electrical and Electronics Engineers (IEEE) spearheaded an effort to standardize system-level interfaces on Unix systems. Richard Stallman, founder of the Free Software movement, suggested the standard be named POSIX (pronounced pahz-icks), which now stands for Portable Operating System Interface.

The first result of this effort, issued in 1988, was IEEE Std 1003.1-1988 (POSIX 1988, for short). In 1990, the IEEE revised the POSIX standard with IEEE Std 1003.1-1990 (POSIX 1990). Optional real-time and threading support were documented in, respectively, IEEE Std 1003.1b-1993 (POSIX 1993 or POSIX.1b), and IEEE Std 1003.1c-1995 (POSIX 1995 or POSIX.1c). In 2001, the optional standards were rolled together with the base POSIX 1990, creating a single standard: IEEE Std 1003.1-2001 (POSIX 2001). The latest revision, released in April 2004, is IEEE Std 1003.1-2004. All of the core POSIX standards are abbreviated POSIX.1, with the 2004 revision being the latest.

In the late 1980s and early 1990s, Unix system vendors were engaged in the "Unix Wars," with each struggling to define its Unix variant as the Unix operating system. Several major Unix vendors rallied around The Open Group, an industry consortium formed from the Open Group early in the 1990s. Specifications such as the hlp standard.

The first step was to mark UNI marked, for Ant system 1003.1-20 when system not ST.

**C Language**


In 1995, the Unix language, ISO C99, became popular, with the introduction of various economic models, and the 1978 publication. POSIX.1: 1978 tried to provide more portability to the C language, with the introduction of the C99 standard.